trapfuzzer:
coverage-guided binary fuzzing with breakpoint

Sili Luo @hac425
About Me

• Security Researcher at Huawei RO0T Lab
• Focus on software vulnerability research
• Github: https://github.com/hac425xxx
• Twitter: https://twitter.com/hac425/
• Blog: https://www.cnblogs.com/hac425
Outline

1. Background

2. Implementation of trapfuzzer

3. How to Fuzz with trapfuzzer and results

4. Future Plans
Background
What is Fuzzing?

def fuzzer(max_length=100, char_start=32, char_range=32):
    """A string of up to `max_length` characters in the range `char_start`, `char_start` + `char_range`"
    string_length = random.randrange(0, max_length + 1)
    out = ""
    for i in range(0, string_length):
        out += chr(random.randrange(char_start, char_start + char_range))
    return out
What is Coverage-Guided Fuzzing?

![Diagram of fuzz testing process]

Fig. 1: fuzz testing process.
Background

and achieves lightweight, low-overhead coverage guided fuzzing for closed source code by:

1. Enumerating the start offset of every basic block in the program/library. This is done with a 
   IDAPython script
2. At runtime, in the fuzzed process, replacing the first byte of every undiscovered basic block
   breakpoint instruction (int3 on Intel). The original byte and the corresponding offset in the c
   bitmap are stored in a dedicated shadow memory mapping whose address can be computed
   address of the modified library, and
3. Installing a [BIGTRAP] handler that will:
   a. Retrieve the faulting address and compute the offset in the library as well as the addi
      corresponding entry in the shadow memory
   b. Mark the basic block as found in the global coverage bitmap
   c. Replace the breakpoint with the original byte
   d. Resume execution

CVE-2020-11784
An out-of-bounds write (of presumably image pixels) on the heap in the copyIntoFramebuffer function.

CVE-2020-11783
A bug that caused a std::vector to be read out-of-bounds. Afterwards, the calling code would write into an
external slice of this vector, thus likely corrupting memory.

CVE-2020-11782
An out-of-bounds memory that was reading out-of-bounds and afterwards potentially writing it out-of-
bounds as well.

CVE-2020-11760, CVE-2020-11781, CVE-2020-11758
Various out-of-bounds reads of pixel data and other data structures.

CVE-2020-11765
An out-of-bounds read on the stack, likely due to an off-by-one error previously overwriting a string null
terminator on the stack.

CVE-2020-11756

https://googleprojectzero.blogspot.com/2020/04/fuzzing-imageio.html

<table>
<thead>
<tr>
<th>Vendor</th>
<th>CVE count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td>27</td>
</tr>
<tr>
<td>Adobe</td>
<td>45</td>
</tr>
<tr>
<td>Apple</td>
<td>2</td>
</tr>
</tbody>
</table>

Inspiration

Combining my previous fuzzing experience and these two security research, I realized:

1. The existing excellent Fuzz tools (AFL, honggfuzz) are not perfect, and there are still unsupported or incompletely supported scenarios, such as complex large, closed-source programs and some new platforms.

2. Using relatively inefficient instrument methods for fuzz testing can also obtain better results than complete black box fuzz testing, such as instruments with breakpoints.

I found that fuzzing tools for large closed-source Linux software are rare, commonly used linux fuzzing tools are Peach and AFL, they have some shortcomings:

1. Peach: fuzzing without coverage information.

2. AFL Qemu Mode: only suitable for relatively small programs, such as image parse library.

So I decided to develop a fuzzer based on a breakpoint mechanism to support some scenarios that are not covered by existing tools, such as large, closed-source file parsing programs and provide coverage support.
Implementation
(Version 0.1)
Overview

Fuzz Scheduler

Seeds

Seed Mutation

Input

Patched Application

Trace Module

Add input to seeds if new path found
binary patcher

1. Use IDAPython script to get all basic blocks of binary
2. Replace the first instruction of every basic blocks with breakpoint instruction and save the original instruction to basic-block-info-file.
binary patcher – basic-block-info-file example

```
```
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Start</th>
<th>Size</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 nva_size</td>
<td>B76000h</td>
<td>0h</td>
<td>4h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct MODULE_NAME</td>
<td>JSTAR024.OCX</td>
<td>4h</td>
<td>11h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>uint32 length</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ubyte data[3]</td>
<td>6h</td>
<td>0h</td>
<td>4h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[0]</td>
<td>id0, nva=0x1000, full0x400, instr size0x1</td>
<td>15h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>uint32 nva</td>
<td>4096</td>
<td>15h</td>
<td>4h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>uint32 slot</td>
<td>1024</td>
<td>19h</td>
<td>4h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>uint32 instr_size</td>
<td>1</td>
<td>10h</td>
<td>4h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>ubyte instr*[1]</td>
<td>21h</td>
<td>1h</td>
<td>1h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[1]</td>
<td>id1, nva=0x1006, full0x400, instr size0x1</td>
<td>22h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[2]</td>
<td>id2, nva=0x1006, full0x400, instr size0x1</td>
<td>25h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[3]</td>
<td>id3, nva=0x1012, full0x400, instr size0x1</td>
<td>32h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[4]</td>
<td>id4, nva=0x1018, full0x400, instr size0x1</td>
<td>49h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[5]</td>
<td>id5, nva=0x101e, full0x400, instr size0x1</td>
<td>56h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[6]</td>
<td>id6, nva=0x1024, full0x400, instr size0x1</td>
<td>63h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
<tr>
<td>struct BB_INFO b[7]</td>
<td>id7, nva=0x102a, full0x400, instr size0x1</td>
<td>70h</td>
<td>0h</td>
<td>fg: 8g</td>
</tr>
</tbody>
</table>
binary patcher - example

breakpoint instruction in x86
binary patcher - example

unpatched

patched

int 3 ; Trap to Debugger

int 3 ; Trap to Debugger
Seed Mutation

RadamsaMutator

use radamsa to generate mutated testcases

https://gitlab.com/akihe/radamsa

TinyMutator

1. support basic mutation strategies, such as byte flipping and random insertion of boundary values (such as 0xFFFFFFFF).

2. supporting config the variation ratio of mutation.
Fuzzer Module

Workflow

1. First load the initial testcases to seed queue and do *corpus distillation* on seed queue.
2. Then it will traverse the seed queue, mutate the testcase, then use the trace module to start the target process, finally get the coverage and execution status of the process.
3. If new coverage is found, the new testcase will be added to the seed queue.
4. If crash is found, the crash information is saved, such as registers, stack trace.

Scheduling strategies

- The score of testcases that discover new coverage will increase ↑
- The score of testcases that trigger DOS will decrease ↓
Fuzzer Module - Corpus Distillation

The workflow of corpus distillation is as follows:
1. Let the program process the testcases in the seed queue one by one.
2. Then only save the testcases that can generate new coverage.
Trace module - Theory

Prep
- Use bb-patcher module.
- Replace the first instruction of basic block with breakpoint instruction (0xcc)

Execution
1. Catch 0xCC exceptions
2. Record location
3. Replace 0xCCC with original value
4. Let process continue
Trace module #1 (PythonPtraceTracer)

1. First use `create_and_attach_process` to create the target process
2. Use `cont` to let the process continue, and use `waitSignals` to wait for the process to trigger signals, such as SIGABORT, SIGTRAP.
3. If the process triggers the `SIGTRAP` signal, record the value of the PC at this time and replace the breakpoint instruction with the original instruction
4. Then let the process continue to execute, `goto 2`
Lets Fuzz WPS

WPS is an office processing software in China that supports viewing and editing DOC, XLS, PPT and other files

- trapfuzzer collect coverage by patch breakpoint instruction to target module
- therefore, it is necessary to find which module need be patched, that is, the module that is responsible for parsing the file
In Windows platform, we can use process monitor to monitor the behavior of the program at runtime, and be able to obtain the call stack, by using this information can quickly locate the data processing module.
Let's Fuzz WPS

What about the Linux platform?

ltrace

No call stack

strace

1. GDB can obtain the call stack stably
2. File operations of the process can be tracked through breakpoints
3. Use GDB's scripting mechanism to automate
class FopenGoodFile(gdb.Breakpoint):
    def __init__(self, name, gdb.BP_BREAKPOINT, internal=False):
        super(FopenGoodFile, self).__init__(name, gdb.BP_BREAKPOINT, internal=False)
        self.hitcount = 0

    def stop(self):
        rdi = read_register("rdi")
        fname = read_memory(rdi, 0x100)
        # print("open: ",fname))
        fname = fname[fname.find(b"\x00")]
        if b"" in fname:
            current_frame = gdb.selected_frame()
            caller = current_frame.older().pc()
            print("[FopenGoodFile] open {}, set bp on fname, caller")
            get_traceback()
            AddBreakf("*{}").format(caller))
            return False
Lets Fuzz WPS

Related modules for processing doc files

```
"coverage_module_name": [
    "libkso.so",
    "libwpsapi.so",
    "libwpsmain.so",
    "wps"
],
```

Linux

```
"coverage_module_name": [
    "wps.exe",
    "wpsmain.dll",
    "kso.dll",
    "ksoapi.dll"
]
```

Windows
Failure and plan

WPS can’t execute within python-trace!

I decided to develop the trace module based on gdb.

The reasons are as follows:

1. Stable, few bugs, support multiple platforms and architectures
2. Support develop plugin with python
3. Open source, can be customized on demand
Implementation

(Version 0.2 - GdbPythonPluginTracer)
GDB Python API

Introduces some commonly used API of GDB Python Plugin

`gdb.selected_frame().read_register()`: read register value
`gdb.selected_inferior().write_memory()`: write process memory
`gdb.selected_inferior().read_memory()`: read process memory
`gdb.parse_and_eval()`: execute gdb expression and get the result of expression
`gdb.execute()`: execute gdb command and get the result of command execution

`gdb.events.stop.connect`: register gdb's stop event callback function, for example, when the process triggers a signal, it will enter the callback function for processing.
Workflow

Due to the limitation of the gdb python script, the fuzzer needs to continuously use stdin and stdout to interact with gdb during the test.
def stop_handler(event):
    if isinstance(event, gdb.StopEvent):
        pc = get_register("$pc") - 1
        hit_mod = None
        for mt in module_trace_list:
            if pc > mt[\"image_base\"] and pc < mt[\"image_end\"]:
                hit_mod = mt
                break

        offset = pc - hit_mod[\"image_base\"]
        raw_byte = get_raw_byte_by_offset(offset)

        # restore original instructions, then set pc pc pc - 1
        write_memory(pc, raw_byte, 1)
        set_register("pc", pc)

        # log executed basic block
        mt[\"bbl-list\"].append(offset)
    else:
        print("Unknown event {}").format(event))

# register callback

gdb.events.stop.connect(stop_handler)
Code – Tracer Part

def exec_with_gdb(self, timeout=30):
    command = "/usr/bin/gdb-q -x {}/cmd.gdb --args {}".format(self.workspace, self.cmdline)
    self.p = subprocess.Popen(command, shell=True, cwd=self.workspace, stdin=subprocess.PIPE,
                               stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    # for dos
    timer = Timer(timeout, self.timeout_handler)
    try:
        timer.start()
        while True:
            l = self.p.stdout.readline()
            # process hit breakpoint
            if "received signal SIGTRAP" in l:
                self.p.stdout.write("c\n")
            # test finished
            elif "[trapfuzzer] save_bb_trace" in l:
                break
        except Exception as e:
            pass
    finally:
        timer.cancel()

self.p.kill()
self.p.wait()
When to kill target process

Problem & Common Solution

WPS will not exit the program after parsing the file, but will stay on the GUI interface and wait for the user to operate, so the fuzzer needs to manually kill the process.

The commonly used method is to set a timeout, when the timeout occurs, the fuzzer will kill the process.

Disadvantage

1. The execution time of the program is fixed each time. It will waste time when process simple testcase (actually parsing time is less than timeout), and when process complex testcase (actually parsing time is greater than timeout), it will result in an incomplete file parsing.
2. Unable to detect DOS vulnerabilities.
When to kill target process - our solution

1. First, use the trace module to trace the program execution and print the basic blocks executed by the program, and find the last basic block END_BBL executed after the process has parsed the input file.

2. In the following fuzzing, when the process reaches END_BBL, it means that the process has entered the GUI loop, and the process can be killed at this time.

During the fuzzing, record the average time (avg_time) of each testcase, and then set the DOS timeout to 10* avg_time.

When the execution time is greater than timeout, it is considered that DOS bug has been found.
When to kill target process
When to kill target process
Speed up Instrument

For large programs, because the number of breakpoints is very large, it will take a lot of time for each execution.

<table>
<thead>
<tr>
<th>Program</th>
<th>desc</th>
<th>basic block count</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPP For Windows</td>
<td>Read and parse PPT file</td>
<td>275 1604</td>
</tr>
<tr>
<td>WPP For Linux</td>
<td>Read and parse PPT file</td>
<td>401 2478</td>
</tr>
<tr>
<td>Ichitaro 2021 Platinum</td>
<td>Read and parse doc, xls and so on</td>
<td>167 3576</td>
</tr>
</tbody>
</table>
Speed up Instrument - accelerated mode

In the accelerated mode, the fuzzer first obtains the executed basic blocks from the trace module, then patch the files related to these basic blocks and remove the breakpoint instructions at the corresponding positions in the files.

At the end of each execution, the executed breakpoint instruction will be removed in the file.
Lets Fuzz WPS Again!
Initial Results

<table>
<thead>
<tr>
<th>Mutator</th>
<th>Crash Count</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadamsaMutator</td>
<td>0</td>
<td>3 * 24h</td>
</tr>
<tr>
<td>TinyMutator</td>
<td>10+</td>
<td>24h</td>
</tr>
</tbody>
</table>

Reason

- RadamsaMutator don’t consider the ratio of data mutation, it may seriously destroy the file structure, causing the testcase to be discarded very early, so that deep vulnerabilities cannot be found.
- TinyMutator can only mutate a small percentage of the data in the sample file, so the use cases can enter a deeper code path and the fuzzing is more efficient.
Implementation

(Version 0.3-trapfuzzer-gdb-tracer)
Why we need trapfuzzer-gdb-tracer

GdbPythonPluginTracer have some limits, such as:
1. It is inconvenient to debug, we need to use a python script to continuously send continue commands to the stdin of gdb.
2. We need to restart gdb for each test, the python runtime, and the communication overhead of fuzzer will cause additional performance overhead.

trapfuzzer-gdb-tracer has following advantages:
1. Written in C++, faster.
2. We can compile static-link GDB to reduce the requirements of environment.
3. Easy to debug.
GDB Internals

On the Linux platform, gdb uses ptrace to debug processes. When gdb starts the debugged program, it will call `start_event_loop` to start the event loop and wait for events from the target process, such as hitting a breakpoint, creating a child process, and so on.

```
start_event_loop () at ../..//gdb/event-loop.c:370
captured_command_loop () at ../..//gdb/main.c:359
captured_main at ../..//gdb/main.c:1202
gdb_main at ../..//gdb/main.c:1217
main at ../..//gdb/gdb.c:32
```
GDB Internals

- `start_event_loop`
- `gdb_do_one_event`
- `fetch_inferior_event`
- `handle_inferior_event`

```c
static void handle_inferior_event(struct execution_control_state *ecs) {
    switch (ecs->ws.kind) {
    case TARGET_WAITKIND_LOADED:
        ............
        ............
    case TARGET_WAITKIND_SPURIOUS:
        ............
        ............
    case TARGET_WAITKIND_STOPPED:
        // handle signal event of target process, such as SIGTRAP
        handle_signal_stop(ecs);
        return;
    }
}
```
GDB Internals

handle_signal_stop handles the situation where the debugged process stops due to receiving a signal,

```c
static void handle_signal_stop(struct execution_control_state *ecs)
{
    frame = get_current_frame();
    gdbarch = get_frame_arch(frame);

    /* Pull the single step breakpoints out of the target. */
    if (ecs->event_thread->suspend.stop_signal == GDB_SIGNAL_TRAP)
    {
        // handle sigtrap signal
    }
```
Modify GDB

Modify the handle_signal_stop function to let gdb automatically handle the breakpoint events of the process:

1. First it gets the value of the pc register, and then finds the module where the pc is located.
2. Then according to pc and basic-block-info-file, get the original instruction of the position.
3. Finally, replace the breakpoint instruction with the original instruction and let the process continue execution.
Modify GDB

We can use add_com to add custom gdb commands

```cpp
C = add_com("load-trapfuzzer-info", class_run, load_trapfuzzer_info, _("Load trapfuzzer config.\nRUN_ARGS_HELP");
set_cmd_completer (c, filename_completer);
```

Use execute_command_to_string to execute gdb command

```cpp
std::string get_context_string()
{
    std::string cmd_res = execute_command_to_string("i r", 0, false);
    cmd_res += execute_command_to_string("x/4i $pc", 0, false);
    cmd_res += execute_command_to_string("bt 4", 0, false);
    return cmd_res;
}
```
Code for SIGTRAP

```c
// handle sigtrap event.
if (ecs->event_thread->suspend.stop_signal == GDB_SIGNAL_TRAP)
{
    pc = regcache_read_pc (regcache);
    // find module of pc
    COV_MOD_INFO* cmi = get_cov_mod_info_by_pc(pc);
    // get module offset
    unsigned int voff = pc - cmi->image_base;
    // remove breakpoint
    BB_INFO* info = cmi->bb_info_map[voff];
    target_write_memory(pc, info->instr, info->instr_size);

    if(g_debug)
        fprintf_unfiltered (gdb_stdlog, "[trapfuzzer] patch to %s!0x%X\n", cmi->module_name, voff);
    // log executed basic block
    cmi->bb_trace.push_back(voff);
    // let process continue.
    keep_going (ecs);
    return;
}
```
Architecture

1. feed mutated input
2. start one test
3. handle the sigtrap in gdb
4. report exec result

Fuzzer

Patched Application

trapfuzzer-gdb
Implementation
(Version 0.4-Windows Support)
Windows Support #1 - winappdbg-tracer

Based on winappdbg

1. based on python.
2. winappdbg basically meets the demand, but there are still some shortcomings: speed, incomplete access to the call stack.
Windows Support #2 – DbgEngTracer

The DbgEng API is a series of APIs for developing debuggers provided by Microsoft. Users only need to register the corresponding callback function to implement a debugger.

Advantage:
1. Fast execution speed.
2. DbgEng API can get the complete call stack of the program.
Windows Support #2 - DbgEngTracer

1. **WriteVirtual/ReadVirtual**: Read/Write process memory.
2. **GetStackTrace**: Get backtrace of process.

```c
EventCallbacks::Exception()
{
  if (Exception->ExceptionCode == STATUS_BREAKPOINT)
  {
    COV_MOD_INFO *cmi = get_cov_mod_info_by_pc(Exception->ExceptionAddress);
    if (cmi != NULL)
    {
      BB_INFO *bi = cmi->bb_info_map[Exception->ExceptionAddress - cmi->image_base];
      printf("exec-bb: %s!%lx\n", cmi->module_name, bi->voff);

      cmi->bb_trace.push_back(bi->voff);
      if (g_Data->WriteVirtual( Exception->ExceptionAddress, bi->instr, 
                                 bi->instr_size, &Done) != S_OK || Done != bi->instr_size)
      {
        return DEBUG_STATUS_NO_CHANGE;
      }
    }
  }
  return DEBUG_STATUS_GO;
}
```
Example
Triage

crash deduplication scheme:
1. first get the call stack of crash
2. then splice the lower 12 bits of each call stack as the hash of crash
3. de-duplicate crash according to hash
Dialog Box

automatically handle dialog box with autoit

```python
def POPUPKillerThread(self):
    while True:
        time.sleep(0.1)

        ppt_text = autoit.win_get_text('Microsoft PowerPoint')

        if u"出现严重错误" in ppt_text:
            autoit.control_click("[Class:#32770]", "Button1")

        if u"无法编辑此" in ppt_text:
            autoit.control_click("[Class:#32770]", "Button1")

        if u"密码" in ppt_text:
            autoit.control_click("[Class:#32770]", "Button1")
```
Dialog Box

automatically handle dialog box with huorong
Other Features

Users can check the status of Fuzz through the management port during the fuzzing process.

nc 127.0.0.1 8821
Start Fuzzing
Three elements of fuzzing

- execution speed
- seed quality/quantity
- mutation algorithm

Ways to obtain samples

1. Obtained from some online sites that provide sample sets
2. Crawl a large number of sample files through the grammar of the search engine
3. Some open source projects will bring some testcases to test the program
4. Testcases generated when using white box Fuzz tools such as AFL to test similar software
5. The bug submission page of the target program or similar program
6. Generated with format conversion tool
Preparing the Environment

Ways for faster execution

use ramdisk, but pay attention to scheduled backups!

Ramdisk in Linux

mount -t tmpfs -o size=4G tmpfs /tmp/ramdisk/

Ramdisk in Windows

ImDisk Toolkit
Equipment & Results

laptop with the following configuration:
i5-8700 (4 Cores 8 threads) / 16G DDR3 RAM / 1T SSD

<table>
<thead>
<tr>
<th>Software</th>
<th>Platform</th>
<th>Time</th>
<th>Initial Seeds</th>
<th>Bug Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPS OFFICE</td>
<td>Linux &amp; Windows</td>
<td>≈ 8 month</td>
<td>Crawl</td>
<td>200+</td>
</tr>
<tr>
<td>中望CAD</td>
<td>Linux</td>
<td>≈ 1 day</td>
<td>AFL (fuzz libredwg)</td>
<td>4+</td>
</tr>
<tr>
<td>WPS Photo</td>
<td>Windows</td>
<td>≈ 1 day</td>
<td>AFL (fuzz libpng)</td>
<td>15</td>
</tr>
<tr>
<td>Foxit PDF</td>
<td>Windows</td>
<td>≈ 7 day</td>
<td>Crawl</td>
<td>a few crashes</td>
</tr>
<tr>
<td>Honeyview</td>
<td>Windows</td>
<td>≈ 1 day</td>
<td>AFL (fuzz libpng)</td>
<td>a few crashes</td>
</tr>
<tr>
<td>Ichitaro 2021</td>
<td>Windows</td>
<td>≈ 7 day</td>
<td>trapfuzzer (fuzz WPS)</td>
<td>100+ unique crash.</td>
</tr>
</tbody>
</table>
## Compare with existing tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Scenarios</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>Library, small program</td>
<td>Coverage, Speed</td>
<td>Large software requires wrapper and high difficulty to user</td>
</tr>
<tr>
<td>Peach</td>
<td>protocol fuzzing, file fuzzing</td>
<td>easy to use after the model is written</td>
<td>model development is difficult</td>
</tr>
<tr>
<td>trapfuzzer</td>
<td>file fuzzing now</td>
<td><strong>easy to use, support large software, support coverage!</strong></td>
<td>Compared with AFL, there is <strong>less feedback coverage and speed is slow.</strong></td>
</tr>
</tbody>
</table>
Conclusion

1. Seed is important and mutate ratio is also important
2. GDB and DbgEng API is very nice.
3. Fuzzing with coverage is much better than none
Future Plans

1. More architecture support (arm, mips)
2. Optimize distributed Fuzz scheduling
3. Automatic detect data mutation ratio and mutation range.
4. More precise crash deduplication mechanism
5. ..................
Q&A

Thank you!
References